2INC0 - Operating Systems

Virtual Memory
Part 2

Geoffrey Nelissen

Interconnected
Resource-aware
Intelligent Systems

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

Course Overview

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduling** (lectures 2+3)
- **Concurrency and synchronization**
  - atomicity and interference (lecture 4)
  - actions synchronization (lecture 5)
  - condition synchronization (lecture 6)
  - deadlock (lecture 7)
- **File systems** (lecture 8)
- **Memory management (lectures 9+10)**
- **Input/output** (lecture 11)

One of the most important subsystems in an OS. It impacts:
- **speed** of execution
- **Maximum size** of executable programs
- **how many** processes can be executed concurrently
- how data and code can be shared
- …

Agenda

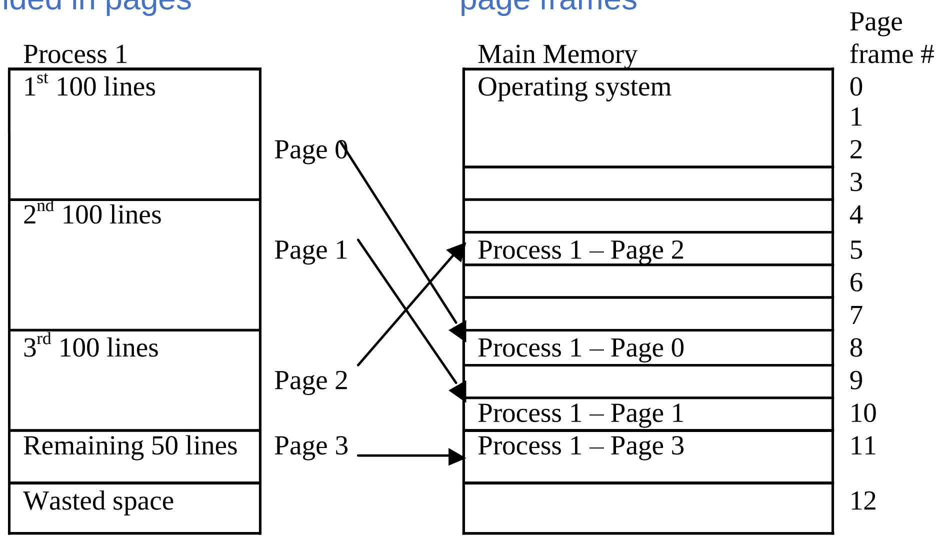
- Reminder from last lecture
- Demand paging
- Page replacement strategies
- Conclusion

Memory paging

Initial objective:
programs do not have to be stored contiguously in main memory

Process address space divided in pages

Physical memory divided in page frames



Advantages
- No external fragmentation
- Limited internal fragmentation
- Only part of the program may be loaded in main memory

Remaining problems:
1. We must **keep track of where pages are stored**
2. We must provide a mechanism for **address binding** (i.e., translate logical addresses into physical addresses)
3. We must decide **which pages** of process **to load and when**

Problem 1: keep track of pages locations in physical memory

Two solutions:

Page table

- **One** page table **per process**
- The page table records, **for each page of the process, in which frame** it is loaded (if it has already been loaded in main memory)
- A pointer to the page table is saved in the PCB of the process

(**For each pallet** in the warehouse, keep track in a ledger **on which shelf each box is** located)

Frame table

- **A single frame table for the whole system**
- The frame table records, **for each frame, which page of which process** is loaded in that frame

(For each shelf in the office, keep track in a ledger which box of which pallet is stored there)

Problem 2: address binding

- Use **virtual addresses** in the program code
- **Translate** virtual addresses in **physical addresses** **during the execution** of the code
- Translation **done with page table or frame table** **by replacing the page number by the frame number** in the virtual address

Where is the data located in the program binary

Virtual address

$|p|$ bits | $|w|$ bits

Page number p	offset

Physical address

$|f|$ bits | $|w|$ bits

Frame number f	offset

Where is the data located in the physical main memory

Virtual memory size seen by each process can be **larger or smaller than** the **physical memory** size

Translation accelerated in hardware using a **TLB** (Translation Look-aside Buffer)

Exercise

Consider a **frame table** with 8 entries as shown below. Each page has a size of 4KiB. The smallest addressable unit is **one byte**.

	Process ID	Page ID
0	5	0x90
1	-	
2	2	0x22
3	2	0x21
4	5	0x80
5	3	0x04
6	1	0x00
7	2	0x04

What is the size of the physical memory?

8 frames of 4KiB ➔ 8*4KiB = 32KiB

On how many bits is the offset of a word encoded?

Pages are 4KiB and we address bytes ➔ 4*1024 addresses in a page ➔ we use 12 bits for the address of a word in a page

What is the physical address for the four following virtual addresses of **process 2**?

0x21650 Physical address: 0x3650

0x80123 Undefined. The page must first be loaded in main memory

0x4341 Physical address: 0x7341

0x20221 Undefined. The page must first be loaded in main memory

Frame table: records **for each frame**, **which page of which process** is loaded in it.

Exercise

Consider the following 8 entries of the **page table** of a process with **8MiB of virtual memory**. Each **page** has a **size of 4KiB**. The smallest addressable unit is a **word of 32 bits**.

	Page table
0	0x5AC0
1	0xFFFF
2	0x1234
3	0xAAAA
4	0xFF
…	
0x11	0x0012
0x12	0xABC
…	
0x123	0xABCD

How many entries is there in the full page table?

8MiB / 4KiB = 2048

How many bits are required to encode the offset of a word?

We have 4KiB/32 bits = 4KiB/4B = 1024 words in a page ➔ we need 10 bits to encode the offset

In what frame can we find the virtual address **0x01234?**

The offset is the last 10 bits of the virtual address (= 0b10 0011 0100 = 0x234), and the page number is the first 10 bits (= 0b00 0000 0100 = 0x4)

➔ The frame in which page 0x4 is loaded is 0xFF (=0b1111 1111)

What is the physical address of the virtual address **0x01234**?

The offset is 0b1000110100 and the frame number is 0b11111111. Concatenating them, we have the physical address 0b11111111 1000110100 = 0x3FE34

Page Table Extensions

Extra fields

Page	Status bit	Referenced bit	Modified bit	Page frame
0	1	1	1	5
1	1	0	0	9
2	1	0	0	7
3	1	1	0	12

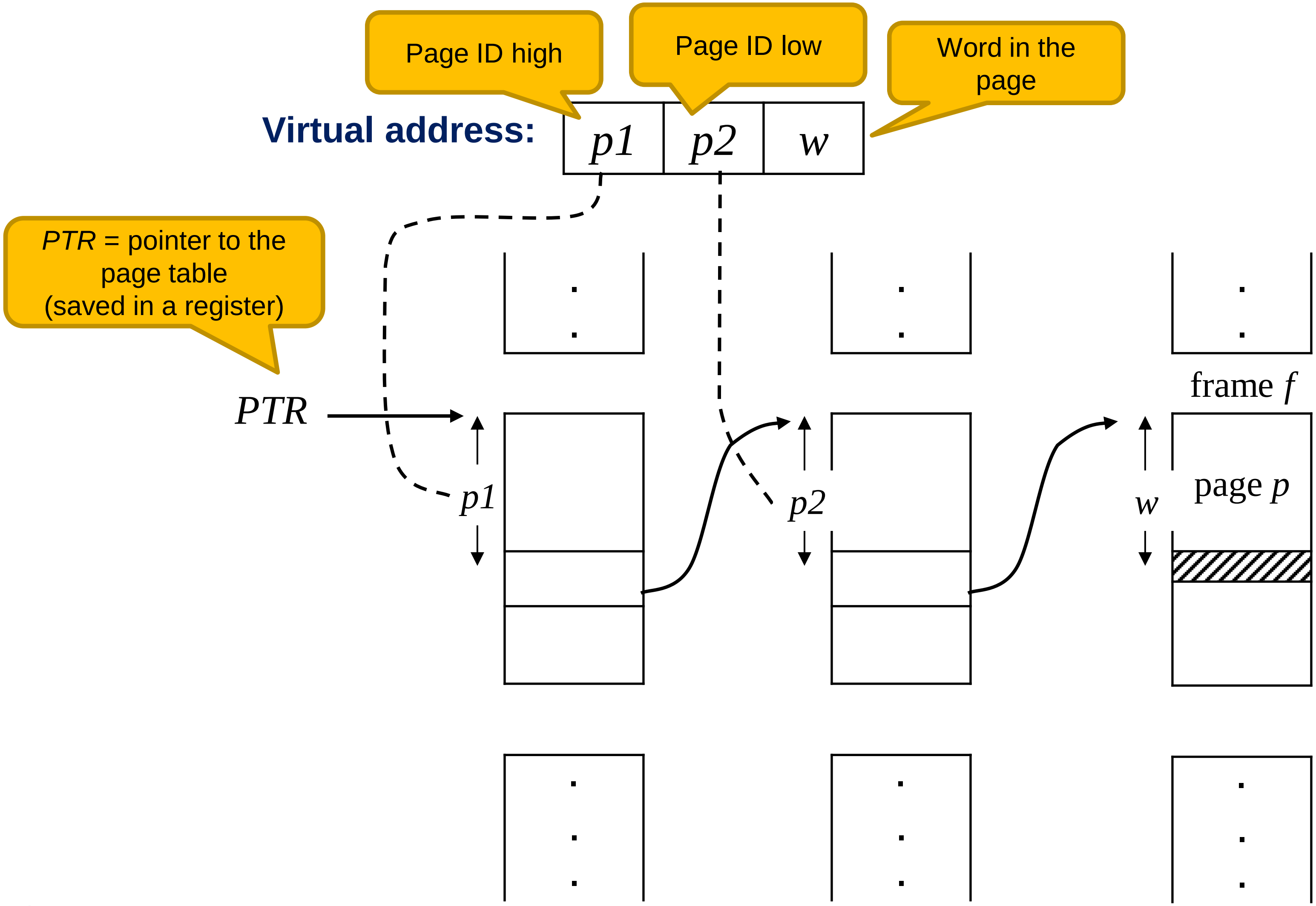
The page table (or frame table) can contain additional bits to keep track of the state of the process pages.

For example:

- **Status bit** indicates whether page is currently in memory or not
- **Referenced bit** (use bit) indicates whether page has been referenced recently
  - Potentially used by replacement policies
- **Modified bit** (dirty bit) indicates whether page contents have been altered
  - Used to determine if page must be written back to secondary storage when it is swapped out
- There may be more of these bits for other purposes, e.g., locking a page in DRAM to avoid it to be swapped out by the replacement algorithm

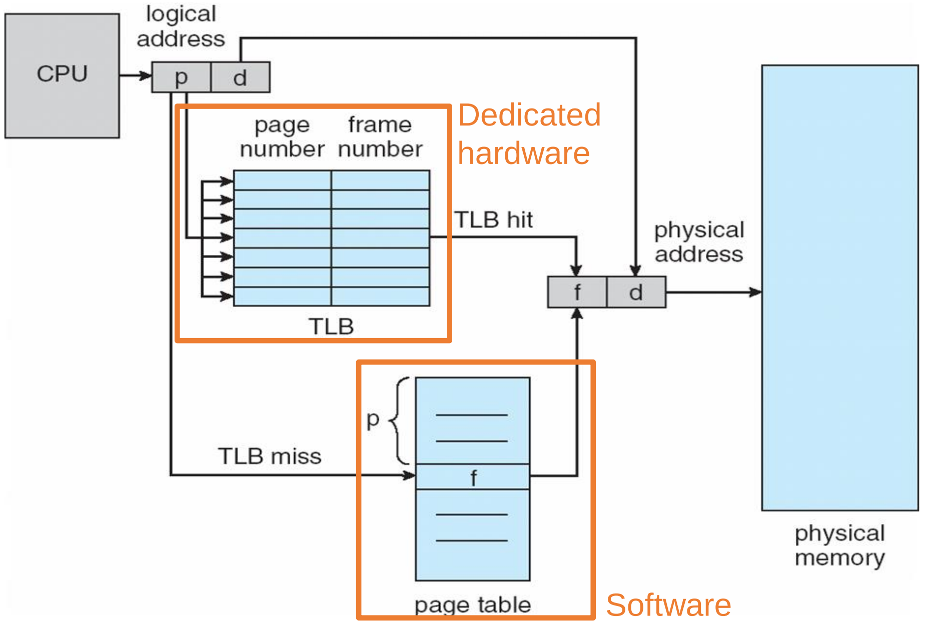
Multi-level page tables

Each **segment is divided in pages**, and the main memory is divided in page frames



Accelerate address binding

- A **Translation Look-aside Buffer** (**TLB**) is a small cache memory in the processor that **keeps track of the locations of the most recently used pages** in main memory
- It **accelerates address translation** and thus memory accesses
- It **does not contain data or instructions**, only the frame id in which the most recently accessed pages are loaded



Source: Fig 9.12, Silberschatz, Galvin, Gagne:Operating System Concepts, 10th Edition, Global edition

- **Whenever a page is accessed**
  - First, check if it is in the TLB
    - **if** the page location **cannot be found in the TLB**, **use the page table**
    - if the page location still cannot be found a **page fault** is generated (i.e., the page is not in physical memory)

Outstanding implementation issues

- **Load control policies**
  - **how many pages** of a process are resident in main memory?
  - **when to load** pages into main memory (demand paging, pre-paging)?

- **Replacement strategies**
  - which page(s) must be **swapped** out **if** there is **not enough free space** in main memory?

- **Sharing**
  - share data and code (e.g., library code) between processes

We do not cover this point in detail in the course
In short, two processes can share access to a subset of pages
➔ code does not have to be duplicated and data can be shared

Agenda

- Reminder
- Demand paging
- Page replacement and load strategies
- Conclusion

Problem 3: avoid loading complete processes

Use **demand paging** instead:

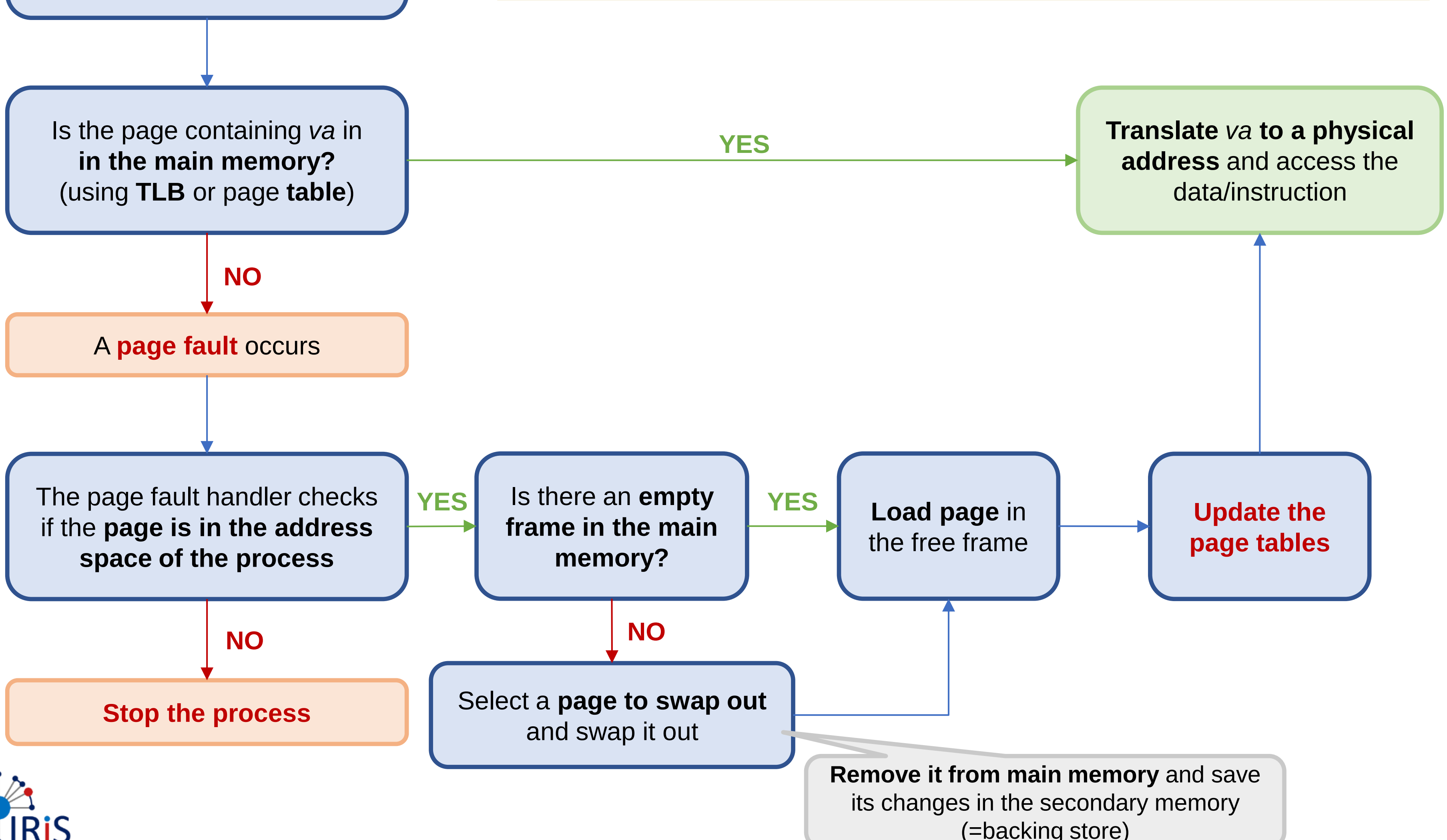
- Bring a **page into main memory** **only when it is needed**
  - No need to have the entire process stored in memory

Takes advantage of the fact that **not all pages are necessary at once**
Examples:
User-written **error handling code**
Mutually exclusive modules/segments of code
Only **fractions of large tables** are actually used

Goal: give the **appearance of an infinite physical memory**

Demand paging: how does it work?

Note: a **single memory reference** may create **several page faults**, e.g., when the page table itself is not in main memory and must be loaded to translate the virtual address in a physical one



Performance of demand paging

- Assume a **page fault rate** $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every memory reference is a page fault

- Our performance metric is the **Effective Access Time** (EAT)

  EAT = (1 – *p*) x memory access time
  + *p* x (page fault overhead + swap page out + swap page in + mem access time)

Example

- **Memory access time** = 200 nanoseconds
- Average **page-fault service time** (overhead+swap out+swap in+mem access) = 8 milliseconds

➔ *EAT* = (1 – *p*) x 200ns + *p* x (8 ms) = 200ns + *p* x 7,999,800ns

➔ If **one access out of 1,000** causes a **page fault**, then ***EAT* = 8.2 microseconds**
This is a **slowdown** by a factor **of 40**!!

- If we want a performance **degradation smaller than 10 percent**

➔ *EAT* < 220ns ⇔ 220ns > 200ns + 7,999,800ns x *p* ⇔ *p* < .0000025

➔ We need **less than one page fault in every 400,000 memory accesses**

Improve performance of demand paging

- Potential solution to reduce the page fault rate:

1. **Pre-paging** in combination with demand paging
  - **Preload pages** of a running process that are **predicted to be accessed soon** in the future, e.g., by loading "super pages" that consist of *x* successive pages of the process at once

2. Keep the **Working Set** in main memory**, i.e., pages that are likely to be reused in the future** (see later)

Relevance of memory paging: Linus Torvald’s comments in Linux

```
// SPDX-License-Identifier: GPL-2.0-only
/*
 *  linux/mm/memory.c
 *
 *  Copyright (C) 1991, 1992, 1993, 1994  Linus Torvalds
 */

/*
 * demand-loading started 01.12.91 - seems it is high on the list of
 * things wanted, and it should be easy to implement. - Linus
 */

/*
 * Ok, demand-loading was easy, shared pages a little bit tricker. Shared
 * pages started 02.12.91, seems to work. - Linus.
 *
 * Tested sharing by executing about 30 /bin/sh: under the old kernel it
 * would have taken more than the 6M I have free, but it worked well as
 * far as I could see.
 *
 * Also corrected some "invalidate()"s - I wasn't doing enough of them.
 */

/*
 * Real VM (paging to/from disk) started 18.12.91. Much more work and
 * thought has to go into this. Oh, well..
 * 19.12.91  -  works, somewhat. Sometimes I get faults, don't know why.
 *              Found it. Everything seems to work now.
```

Source: https://github.com/torvalds/linux

Agenda

- Reminder
- Demand paging
- Page replacement and load strategies
- Conclusion

Thrashing

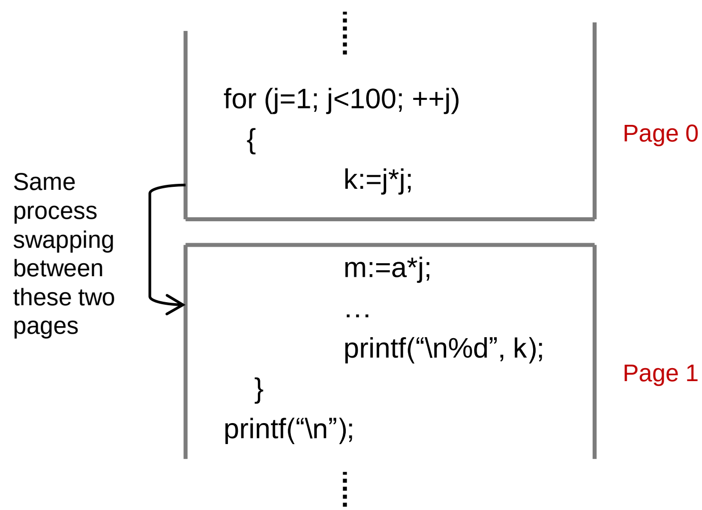
On a page fault, how do we decide **what page to swap out?**

Note that a **wrong replacement strategy** could **result in "trashing"**

Thrashing

Pages in active use are replaced by other pages in active use resulting in swapping pages in main memory too often
➔ a process spends more time treating page faults than executing

- **Examples:**
  1. Two **different processes** **compete for the same frame**
  2. Two pages of the **same process** compete for the same frame



Load and replacement strategies

Two problems:

- Physical memory has a **limited number of frames**.
  How many should be allocated to **each active process**?
- If a process must load a new page and there is **no free frame**, **which page do we swap out** from main memory?

Several options:

1. All processes **compete for all frames**
   - If a page must be swapped out, it can be from any process
2. Each active process is assigned a **fixed number of frames** based on some criteria (e.g., proportional to its virtual address space size, its priority, …)
   - Two options when selecting a page to swap out:
     - It can only be a page from the process loading a new page
     - It can be a page from the process loading a new page, or any lower priority process
3. Use the ***"working set"*** approach (see later)

Load and replacement strategies

Two problems:

- Physical memory has a **limited number of frames**.
  How many should be allocated to **each active process**?
- If a process must load a new page and there is **no free frame**, **which page do we swap out** from main memory?

Several options:

1. All processes **compete for all frames**
   - If a page must be swapped out, it can be from any process
2. Each active process is assigned a **fixed number of frames** based on some criteria (e.g., proportional to its address space size, its priority, …)
   - Two options when selecting a page to swap out:
     - It can only be a page from the process loading a new page
     - It can be a page from the process loading a new page, or any lower priority process
3. Use the ***"working set"*** approach (see later)

Agenda

- Reminder
- Memory paging
- Page replacement and load strategies
  - Global
  - Working set
- Conclusion

Global replacement strategies

When we must swap out a page, select any page in main memory according to one of the following strategies:

Theoretical algorithm that minimize the number of page faults

- **MIN** replacement (**looks to the future**)
  - select the **page which will not be used for the longest time in the future**
    ➔ this gives the minimum number of page faults
- **Random** replacement
  - select a **random page** for replacement
- **FIFO** replacement
  - select the page that has been resident **in main memory for the longest time**
- **LRU** replacement
  - select the page that is **least recently used**
- Clock replacement (**second chance**) 🡐See reference book for an example
  - **circular list** of all resident pages equipped **with a use-bit** u
  - **upon each reference** u **is set to 1**
  - **search clockwise for the first page with** $u=0$**, while setting the use-bits to zero**

Comparison of replacement strategies is done **using reference strings**
- i.e., an **execution trace** in which only **memory references** are recorded
- only the **page number** of the referenced location is mentioned

Goodness criteria: the number of generated page faults

MIN policy (looks to the future)

MIN replacement (looks to the future)
select the page which will not be used for the longest time in the future

Theoretical algorithm. Impossible to implement in a real system.

Reference string:	A	B	A	C	A	B	D	B	A	C	D
Page Frame 1 (empty)											
Page Frame 2 (empty)											
Time:	1	2	3	4	5	6	7	8	9	10	11

- **11 page requests** are issued.
- **How many page faults** are generated?

MIN policy (looks to the future)

MIN replacement (looks to the future)
select the page which will not be used for the longest time in the future

Theoretical algorithm. Impossible to implement in a real system.

Reference string:	A	B	A	C	A	B	D	B	A	C	D
	Page Frame 1 **A**	A	A	A	A	A	**D**	D	D	D	D
	Page Frame 2 (empty)	**B**	B	**C**	C	**B**	B	B	**A**	**C**	C
Page fault	*	*		*		*	*		*	*	
Time:	1	2	3	4	5	6	7	8	9	10	11

- **11 page requests** are issued.
- **7 page faults** are generated

LRU policy

LRU replacement
select the page that is least recently used

The **most widely used** replacement algorithm

Reference string:	A	B	A	C	A	B	D	B	A	C	D
Time:	1	2	3	4	5	6	7	8	9	10	11

Page Frame 1
(empty)

Page Frame 2
(empty)

- **11 page requests** are issued.
- **How many page faults** are generated?

LRU policy

LRU replacement
select the page that is least recently used

The **most widely used** replacement algorithm

Page Requested:	A	B	A	C	A	B	D	B	A	C	D
Page Frame 1	Page A	A	A	A	A	A	D	D	A	A	D
Page Frame 2	(empty)	B	B	C	C	B	B	B	B	C	C
Page fault	*	*		*		*	*		*	*	*
Time:	1	2	3	4	5	6	7	8	9	10	11

- Only **8 page faults** generated

FIFO policy

FIFO replacement
select the page that has been resident in main memory for the longest time

Page Requested:	A	B	A	C	A	B	D	B	A	C	D
Page Frame 1	Page A	A	A	C	C	B	B	B	A	A	D
Page Frame 2	(empty)	B	B	B	A	A	D	D	D	C	C
Page fault	*	*		*	*	*	*		*	*	*
Time:	1	2	3	4	5	6	7	8	9	10	11

- **9 page faults** generated

Agenda

- Reminder
- Memory paging
- Page replacement and load strategies
  - Global
  - Working set
- Conclusion

Working set

- It uses the **time locality** property of programs, i.e., the set of pages accessed by a program remains rather constant on short periods of time
- **Keep** only the **pages of the past τ memory references** made by each process **in main memory**

- The working set at time t is given by $W(t, \tau) = \{ r_j \mid t-\tau < j \leq t \}$ for a reference string: $r_0 r_1 r_2 \ldots r_T$

Example

Reference string for process P: …2 6 1 5 7 7 7 7 5 1 | 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 | 2 3 4 4 4…

t_1 t_2

What is the **working set at time t1 and t2** assuming a window length $\tau = 10$?

Working set

- It uses the **time locality** property of programs, i.e., the set of pages accessed by a program remains rather constant on short periods of time
- **Keep** only the **pages of the past τ memory references** made by each process **in main memory**

- The working set at time t is given by $W(t, \tau) = \{ r_j \mid t-\tau < j \leq t \}$ for a reference string: $r_0 r_1 r_2 \ldots r_T$

Example

Reference string for process P : …2 6 1 5 7 7 7 7 5 1 | 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 | 2 3 4 4 4…

$\tau = 10$

t_1 t_2

WS (t_1)={1,2,5,6,7}

Working set

- It uses the **time locality** property of programs, i.e., the set of pages accessed by a program remains rather constant on short periods of time
- **Keep** only the **pages of the past τ memory references** made by each process **in main memory**

- The working set at time t is given by $W(t, \tau) = \{ r_j \mid t-\tau < j \leq t \}$ for a reference string: $r_0 r_1 r_2 \ldots r_T$

Example

Reference string for process P : …2 6 1 5 7 7 7 7 5 1 | 6 2 3 4 1 2 3 4 | 4 4 3 4 3 4 4 4 1 3 | 2 3 4 4 4…

$\tau = 10$ (ending at t_1), $\tau = 10$ (ending at t_2)

WS (t_1)={1,2,5,6,7}
WS (t_2)={1,3,4}

Working set size ($WSS_i(t)$) of process *i* at time *t* is the **number of pages in its working set** at time *t*

- If τ **is too small**, there will be **thrashing**
- If τ **is too large**, **less processes** can fit in main memory

Working set –an example-

-2	-1	Time t	0	1	2	3	4	5	6	7	8	9	10
e	d	Reference string	a	c	c	d	b	c	e	c	e	a	d
		Page a	√	√	√	√	--	--	--	--	--	√	√
		Page b	--	--	--	--	√	√	√	√	--	--	--
		Page c	--	√	√	√	√	√	√	√	√	√	√
		Page d	√	√	√	√	√	√	√	--	--	--	√
		Page e	√	√	--	--	--	--	√	√	√	√	√
		IN_t	a*	c *			b *		e *			a *	d *
		OUT_t			e		a			d	b		

Evolution of the working set of one process **for τ=4**

Load control

- If the **sum of the working sets sizes** , i.e, $\sum_i WSS_i(t)$, **is larger than the main memory size,** then **remove one process** from main memory

Tries to avoid trashing between processes

- Several options to **choose a victim**
  - **lowest priority** process
    - follows CPU scheduling (unlikely to be immediately scheduled again)
  - **last process activated**
    - considered to be the least important
  - **smallest process**
    - least expensive to swap out
  - **largest process**
    - frees the largest number of page frames

Agenda

- Reminder
- Memory paging
- Page replacement and load strategies
- Segmentation
- Conclusion

Virtual memory

Advantages:

- **Process size is no longer restricted to main memory size** (or the free space within main memory)
- Memory is used more efficiently
  - **Eliminates external fragmentation** when used with paging
  - **Reduces internal fragmentation**
- Placement of a program in memory does not have to be known at design time
- Facilitates dynamic linking of program segments
- **Allows sharing of code and data (by sharing access to pages)**

Disadvantages:

- Increased **processor hardware costs**
- Increased **overhead** for handling page faults
- **Increased software complexity to prevent thrashing**

Summary

Simple	Yes	User has access to a linear address space
Private	Yes	Virtual memory also facilitates sharing
Permanent	No	Unless the programmer enforces this during execution.
Fast	Moderate	Management overhead for tables and replacement strategies but **HW support helps accelerate memory management**
Huge	Yes	Memory size virtually unlimited
Cost-effective	Yes	Thanks to the memory hierarchy, but hardware support for virtual memory can be moderately expensive

What next?

- **Exercises available** on Canvas
  - Do them, to **prepare for the exam**.

- **Two more homeworks**

- **One lecture left**:
  - I/O management

Enjoy your winter break!